University of Castilla-La Mancha

A publication of the

Department of Computer Science

Automatic Tupling for Functional-Logic Programs

by

Ginés Moreno

Technical Report **#DIAB-02-07-24** May 2002

DEPARTAMENTO DE INFORMÁTICA
ESCUELA POLITÉCNICA SUPERIOR
UNIVERSIDAD DE CASTILLA-LA MANCHA
Campus Universitario s/n
Albacete - 02071 - Spain
Phone +34.967.599200, Fax +34.967.599224

Automatic Tupling for Functional-Logic Programs ⋆

Ginés Moreno

Dep. Informática, UCLM, 02071 Albacete, Spain. gmoreno@info-ab.uclm.es

Abstract. This paper investigates the optimization by fold/unfold of functional-logic programs with operational semantics based on needed narrowing. Transformation sequences are guided by the tupling strategy, a powerful method that avoids multiple accesses to data structures and redundant sub-computations. We systematically decompose the internal structure of tupling in three low-level transformation phases in order to built a global automatic algorithm where eureka definitions are generated in an incremental way. The resulting strategy is (strongly) correct and complete, it avoids redundant transformation steps and it is elegant, realistic and applicable to most practical cases. Moreover and most important, our technique is specially well-suited for functional-logic programs since, in contrast with prior approaches for pure functional programs that may produce corrupt integrated programs, (forbidden) extra variables on rules are avoided and non overlapping rules are produced.

Keywords: Functional logic programming, fold/unfold, tupling

1 Introduction

Functional logic programming languages combine the operational methods and advantages of the most important declarative programming paradigms, namely functional and logic programming. The operational principle of such languages is usually based on *narrowing*. A *narrowing step* instantiates variables in an expression and applies a reduction step to a redex of the instantiated expression. Needed narrowing is the currently best narrowing strategy for first-order (inductively sequential) functional logic programs due to its optimality properties w.r.t. the length of derivations and the number of computed solutions [6], and it can be efficiently implemented by pattern matching and unification.

The fold/unfold transformation approach was first introduced in [10] to optimize functional programs and then used for logic programs [22]. This approach is commonly based on the construction, by means of a *strategy*, of a sequence of equivalent programs each obtained from the preceding ones by using an *elementary* transformation rule. The essential rules are *folding* and *unfolding*,

⋆ This work has been partially supported by CICYT under grant TIC 2001-2705-C03-03 and by Acción Integrada Hispano-Italiana HI2000-0161.

i.e., contraction and expansion of subexpressions of a program using the definitions of this program (or of a preceding one). Other rules which have been considered are, for example: instantiation, definition introduction/elimination, and abstraction. The first attempt to introduce these ideas in an integrated language is presented in [2], where we investigated fold/unfold rules in the context of a strict (*call-by-value*) functional logic language. A transformation methodology for lazy (*call-by-name*) functional logic programs was introduced in [3]; this work extends the transformation rules of [22] for logic programs in order to cope with lazy functional logic programs (based on needed narrowing). The use of narrowing empowers the fold/unfold system by implicitly embedding the instantiation rule (the operation of the Burstall and Darlington framework [10] which introduces an instance of an existing equation) into unfolding by means of unification. [3] also demonstrates that the (inductively sequential) structure of programs is preserved through the transformation sequence $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$, which is a key point for proving the correctness of the transformation system as well as for being effectively applicable.

There exists a large class of program optimizations which can be achieved by fold/unfold transformations and are not possible by using a fully automatic method (such as, e.g., partial evaluation). Typical instances of this class are the strategies that perform *tupling* (also known as *pairing*) [10, 13], which merges separate (nonnested) function calls with some common arguments into a single call to a (possibly new) recursive function which returns a tuple of the results of the separate calls, thus avoiding either multiple accesses to the same data structures or common subcomputations, similarly to the idea of *sharing* which is used in graph rewriting to improve the efficiency of computations in time and space [7].

However, a weak point of the fold/unfold transformation approach is the achievement of the appropriate set of (*eureka*) definitions which make it possible the optimizations to be pursued [10, 22, 20]. In this paper, we propose a fully automatic tupling algorithm (based on folding/unfolding) which uses an incremental eureka generator. In contrast with prior non-automatic approaches (tupling has only been semi-automated to some extent [11, 12]) where eurekas are generated by a complicate pre-process that uses complex data structures and/or produces redundant computations, our approach is fully automatic and covers most practical cases. The key point of our algorithm is a simple eureka generator that, in an incremental way, produces an optimal set of eureka definitions, that is, eureka generation is done simultaneously at transformation time. By recursive calls to the eureka generator and the general tupling algorithm, we obtain new eureka definitions more and more refined, until reaching the intended optimization. Our method deals with particular (non trivial) features of integrated (functional-logic) languages and includes refined tests for termination of each transformation phase. More exactly, we have identified three syntactic conditions to stop the search for regularities during the unfolding phase[1].

[1] The method is not universal but, as said in [20], "one cannot hope to construct an universal technique for finding a suitable regularity whenever there is one".

The structure of the paper is as follows. After recalling some basic definitions in the next section, we introduce the basic transformation rules and illustrate its use by means of simple tupling examples in Section 3. Then, we describe in Section 4 the three different transformation phases that constitute the core of an elemental tupling algorithm. Section 5 shows that these building blocks can be chained in a more refined way in order to build an incremental (recursive) algorithm that performs automatic tupling. We discute and compare our method with related works and we also illustrate the benefits of our approach (gains in efficiency, preservation of program structure, etc.) by means of several examples. Finally, Section 6 concludes.

2 Preliminaries

We assume familiarity with basic notions from term rewriting [14] and functional logic programming [15]. In this work we consider a (*many-sorted*) *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of *defined* functions. The set of *constructor terms* with *variables* (e.g., x, y, z) is obtained by using symbols from $\mathcal{C}$ and $\mathcal{X}$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. We write $\overline{o_n}$ for the *list* of objects $o_1, \ldots, o_n$. A *pattern* is a term of the form $f(\overline{d_n})$ where $f/n \in \mathcal{F}$ and $d_1, \ldots, d_n$ are constructor terms. A term is *linear* if it does not contain multiple occurrences of one variable. A term is *operation-rooted* (*constructor-rooted*) if it has an operation (constructor) symbol at the root. A *position* p in a term t is represented by a sequence of natural numbers. (Λ denotes the empty sequence, i.e., the root position). Positions are ordered by the *prefix* ordering: $p \leq q$, if $\exists w$ such that $p.w = q$. Positions p, q are *disjoint* if neither $p \leq q$ nor $q \leq p$. Given a term t, we let $\mathcal{P}os(t)$ and $\mathcal{FP}os(t)$ denote the set of positions and the set of non-variable positions of t, respectively. $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s. For a sequence of (pairwise disjoint) positions $P = \overline{p_n}$, we let $t[\overline{s_n}]_P = (((t[s_1]_{p_1})[s_2]_{p_2}) \ldots [s_n]_{p_n})$. By abuse, we denote $t[\overline{s_n}]_P$ by $t[s]_P$ when $s_1 = \ldots = s_n = s$, as well as $((t[s_1]_{P_1}) \ldots [s_n]_{P_n})$ by $t[\overline{s_n}]_{\overline{P_n}}$. We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ with $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $\sigma(x) = x$ for all other variables x. *id* denotes the identity substitution.

A set of rewrite rules $l \to r$ such that $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ is called a *term rewriting system* (TRS). The terms l and r are called the *left-hand side* (lhs) and the *right-hand side* (rhs) of the rule, respectively. A TRS $\mathcal{R}$ is left-linear if l is linear for all $l \to r \in \mathcal{R}$. A TRS is constructor–based (CB) if each left-hand side is a pattern. In the remainder of this paper, a functional logic *program* is a left-linear CB-TRS. A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there exists a position p in t, a rewrite rule $R = (l \to r)$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. Given a relation $\to$, we denote by $\to^+$ its transitive closure and by $\to^*$ its reflexive and transitive closure.

The operational semantics of integrated languages is usually based on *narrowing*, a combination of variable instantiation and reduction. Formally, $s \leadsto_{p,R,\sigma} t$

is a *narrowing step* if p is a non-variable position in s and $\sigma(s) \to_{p,R} t$. We denote by $t_0 \leadsto_\sigma^* t_n$ a sequence of narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$). Modern functional logic languages are based on *needed narrowing* and *inductively sequential* programs.

Needed Narrowing. A challenge in the design of functional logic languages is the definition of a "good" narrowing strategy, i.e., a restriction λ on the narrowing steps issuing from t, without losing completeness. *Needed narrowing* [6] is currently the best known narrowing strategy due to its optimality properties. It extends the Huet and Lévy's notion of a needed reduction [18]. The definition of needed narrowing [6] uses the notion of *definitional tree* [5]. Roughly speaking, a definitional tree for a function symbol f is a tree whose leaves contain all (and only) the rules used to define f and whose inner nodes contain information to guide the (optimal) pattern matching during the evaluation of expressions. Each inner node contains a pattern and a variable position in this pattern (the *inductive position*) which is further refined in the patterns of its immediate children by using different constructor symbols. The pattern of the root node is simply $f(\overline{x_n})$, where $\overline{x_n}$ are different variables. A defined function is called *inductively sequential* if it has a definitional tree. A rewrite system $\mathcal{R}$ is called *inductively sequential* if all its defined functions are inductively sequential.

To compute needed narrowing steps for an operation-rooted term t, we take a definitional tree $\mathcal{P}$ for the root of t and compute $\lambda(t, \mathcal{P})$. Then, for all $(p, R, \sigma) \in \lambda(t, \mathcal{P})$, $t \leadsto_{p,R,\sigma} t'$ is a *needed narrowing step*. Informally speaking, needed narrowing applies a rule, if possible, or checks the subterm corresponding to the inductive position of the branch: if it is a variable, it is instantiated to the constructor of a child; if it is already a constructor, we proceed with the corresponding child; if it is a function, we evaluate it by recursively applying needed narrowing (see [6] for a detailed definition).

Example 1. Consider the following set of rules for "$\leqslant$" and "+":

$$
\begin{array}{rclcl@{\qquad}rcl}
\mathtt{0} & \leqslant & \mathtt{N} & \to & \mathtt{true} & \mathtt{0} + \mathtt{N} & \to & \mathtt{N} \\
\mathtt{s(M)} & \leqslant & \mathtt{0} & \to & \mathtt{false} & \mathtt{s(M)} + \mathtt{N} & \to & \mathtt{s(M + N)} \\
\mathtt{s(M)} & \leqslant & \mathtt{s(N)} & \to & \mathtt{M} \leqslant \mathtt{N} & & &
\end{array}
$$

Then the function λ computes the following set for the initial term $\mathtt{X} \leqslant \mathtt{X} + \mathtt{X}$:

$$\{(\Lambda, \mathtt{0} \leqslant \mathtt{N} \to \mathtt{true}, \{\mathtt{X} \mapsto \mathtt{0}\}),\ (2, \mathtt{s(M)} + \mathtt{N} \to \mathtt{s(M+N)}, \{\mathtt{X} \mapsto \mathtt{s(M)}\})\}$$

This corresponds to the narrowing steps (the subterm evaluated in the next step is underlined):

$$
\begin{array}{l}
\underline{\mathtt{X} \leqslant \mathtt{X} + \mathtt{X}} \leadsto_{\{\mathtt{X} \mapsto \mathtt{0}\}} \mathtt{true} \\
\mathtt{X} \leqslant \underline{\mathtt{X} + \mathtt{X}} \leadsto_{\{\mathtt{X} \mapsto \mathtt{s(M)}\}} \mathtt{s(M)} \leqslant \mathtt{s(M + s(M))}
\end{array}
$$

Needed narrowing is sound and complete for inductively sequential programs. Moreover, it is *minimal*, i.e., given two distinct needed narrowing derivations $e \leadsto_\sigma^* true$ and $e \leadsto_{\sigma'}^* true$, we have that σ and σ' are independent on $\mathcal{V}ar(e)$.

3 Fold/Unfold Transformation Sequences

Firstly, we recall from [3] the basic definitions of the transformation rules. Programs in a transformation sequence constructed by using the following set of rules are inductively sequential. Moreover, the transformations are strongly correct w.r.t. goals containing (*old*) function symbols from the initial program.

Definition 1. *Let $\mathcal{R}_0$ be an inductively sequential program (the original program). A* transformation sequence *$(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $k > 0$ is constructed by applying the following transformation rules:*

Definition Introduction: *We may get program $\mathcal{R}_{k+1}$ by adding to $\mathcal{R}_k$ a new rule ("definition rule" or "eureka") of the form $f(\overline{x}) \to r$, where f is a* new *function symbol not occurring in the sequence $\mathcal{R}_0, \ldots, \mathcal{R}_k$ and $\mathcal{V}ar(r) = \overline{x}$.*

Unfolding: *Let $R = (l \to r) \in \mathcal{R}_k$ be a rule where r is an operation-rooted term or $\mathcal{R}_k$ is completely defined. Then, $\mathcal{R}_{k+1} = (\mathcal{R}_k - \{R\}) \cup \{\theta(l) \to r' \mid r \leadsto_\theta r' \text{ in } \mathcal{R}_k\}$. An unfolding step where $\theta = id$ is called a normalizing step.*

Folding: *Let $R = (l \to r) \in \mathcal{R}_k$ be a non definition rule, $R' = (l' \to r') \in \mathcal{R}_j$, $0 \leq j \leq k$, a definition rule[2] and p a position in r such that $r|_p = \theta(r')$ and $r|_p$ is not a constructor term. Then, $\mathcal{R}_{k+1} = (\mathcal{R}_k - \{R\}) \cup \{l \to r[\theta(l')]_p\}$.*

Abstraction: *Let $R = (l \to r) \in \mathcal{R}_k$ be a rule and let $\overline{P_j}$ be sequences of disjoint positions in $\mathcal{FP}os(r)$ such that $r|_p = e_i$ for all p in P_i, $i = 1, \ldots, j$, i.e., $r = r[\overline{e_j}]_{\overline{P_j}}$. We may get program $\mathcal{R}_{k+1}$ from $\mathcal{R}_k$ by replacing R with $l \to r[\overline{z_j}]_{\overline{P_j}}$* where *$\langle z_1, \ldots, z_j \rangle = \langle e_1, \ldots, e_j \rangle$ (where $\overline{z_j}$ are fresh variables not occurring in l).*

The following well-known example uses the previous set of transformation rules for optimizing a program following a tupling strategy. The process is similar to [10, 11, 21] for pure functional logic programs, with the advantage in our case that we avoid the use of an explicit *instantiation rule* before applying unfolding steps. As pointed out in [3] this is possible thanks to the systematic instantiation of calls performed implicitly by our unfolding rule (by virtue of the logic component of the needed narrowing mechanism).

Example 2. The fibonacci numbers can be computed by the program $\mathcal{R}_0$:

(R_1) $\quad \texttt{fib(0)} \to \texttt{s(0)}$
(R_2) $\quad \texttt{fib(s(0))} \to \texttt{s(0)}$
(R_3) $\quad \texttt{fib(s(s(X)))} \to \texttt{fib(s(X))} + \texttt{fib(X)}$

(together with the rules for addition $+$). Observe that this program has an exponential complexity, which can be reduced to a linear one by applying the tupling strategy as follows:

[2] A *definition rule* (*eureka*) maintains its status only as long as it remains unchanged, i.e., once a definition rule is transformed it is not considered a *definition rule* anymore.

1. Definition introduction:
   $(R_4)\quad \texttt{new(X)} \to \langle \texttt{fib(s(X))}, \texttt{fib(X)} \rangle$
2. Unfolding of rule R_4 (narrowing the needed redex $\texttt{fib(s(X))}$):
   $(R_5)\quad \texttt{new(0)} \to \langle \texttt{s(0)}, \texttt{fib(0)} \rangle$
   $(R_6)\quad \texttt{new(s(X))} \to \langle \texttt{fib(s(X))} + \texttt{fib(X)}, \texttt{fib(s(X))} \rangle$
3. Unfolding (normalizing) of rule R_5 (reducing subterm $\texttt{fib(0)}$):
   $(R_7)\quad \texttt{new(0)} \to \langle \texttt{s(0)}, \texttt{s(0)} \rangle$
4. Abstraction of R_6:
   $(R_8)\quad \texttt{new(s(X))} \to \langle \texttt{Z}_1 + \texttt{Z}_2, \texttt{Z}_1 \rangle \ \texttt{where} \ \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \langle \texttt{fib(s(X))}, \texttt{fib(X)} \rangle$
5. Folding of $\langle \texttt{fib(s(X))}, \texttt{fib(X)} \rangle$ in rule R_8 using R_4:
   $(R_9)\quad \texttt{new(s(X))} \to \langle \texttt{Z}_1 + \texttt{Z}_2, \texttt{Z}_1 \rangle \ \texttt{where} \ \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \texttt{new(X)}$
6. Abstraction of R_3:
   $(R_{10})\quad \texttt{fib(s(s(X)))} \to \texttt{Z}_1 + \texttt{Z}_2 \ \texttt{where} \ \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \langle \texttt{fib(s(X))}, \texttt{fib(X)} \rangle$
7. Folding of $\langle \texttt{fib(s(X))}, \texttt{fib(X)} \rangle$ in rule R_{10} using again rule R_4:
   $(R_{11})\quad \texttt{fib(s(s(X)))} \to \texttt{Z}_1 + \texttt{Z}_2 \ \texttt{where} \ \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \texttt{new(X)}$

Now, the (enhanced) transformed program $\mathcal{R}_7$ (with linear complexity thanks to the use of the recursive function `new`), is the following:

$$
\begin{array}{lrl}
(R_1) & \texttt{fib(0)} & \to \texttt{s(0)} \\
(R_2) & \texttt{fib(s(0))} & \to \texttt{s(0)} \\
(R_{11}) & \texttt{fib(s(s(X)))} & \to \texttt{Z}_1 + \texttt{Z}_2 \ \texttt{where} \ \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \texttt{new(X)} \\
(R_7) & \texttt{new(0)} & \to \langle \texttt{s(0)}, \texttt{s(0)} \rangle \\
(R_9) & \texttt{new(s(X))} & \to \langle \texttt{Z}_1 + \texttt{Z}_2, \texttt{Z}_1 \rangle \ \texttt{where} \ \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \texttt{new(X)}
\end{array}
$$

The classical instantiation rule used in pure functional transformation systems is problematic since it uses most general unifiers of expressions (it is commonly called "minimal instantiation" [11]) and, for that reason, it is rarely considered explicitly in the literature. Moreover, in a functional-logic setting the use of an unfolding rule that (implicitly) performs minimal instantiation may generate corrupt programs that could not be executed by needed narrowing. The reader must note the importance of this fact, since it directly implies that tupling algorithms (that performs minimal instantiation) developed for pure functional programs are not applicable in our framework, as illustrates the following example.

Example 3. Consider a functional-logic program containing (among others) the following well-known set of non overlapping rules:

$$
\begin{array}{rlcrl}
\texttt{leq(0,X)} & \to \texttt{true} & \quad & \texttt{double(0)} & \to \texttt{0} \\
\texttt{leq(s(X),0)} & \to \texttt{false} & & \texttt{double(s(X))} & \to \texttt{s(s(double(X)))} \\
\texttt{leq(s(X),s(Y))} & \to \texttt{leq(X,Y)} & & \ldots\ldots &
\end{array}
$$

Assume now that a tupling strategy is started and after some definition introduction and unfolding steps we obtain the following new rule:

$$\texttt{new}(\ldots, \texttt{X}, \texttt{Y}, \ldots) \to \langle \ldots, \underline{\texttt{leq(X,double(Y))}}, \ldots \rangle$$

Now, if we apply an unfolding step over the underlined term with implicitly performs minimal instantiation before (lazily) reducing it, we obtain:

$$\begin{aligned}
\texttt{new}(\ldots,\texttt{0},\texttt{Y},\ldots) &\to \langle\ldots,\texttt{true},\ldots\rangle \\
\texttt{new}(\ldots,\texttt{X},\texttt{0},\ldots) &\to \langle\ldots,\texttt{leq}(\texttt{X},\texttt{0}),\ldots\rangle \\
\texttt{new}(\ldots,\texttt{X},\texttt{s}(\texttt{Y}),\ldots) &\to \langle\ldots,\texttt{leq}(\texttt{X},\texttt{s}(\texttt{s}(\texttt{double}(\texttt{Y})))),\ldots\rangle
\end{aligned}$$

And now observe that there exist overlapping rules that loose the program structure. This loss prevents for further computations with needed narrowing. Fortunately (as proved in [3]), our unfolding rule always generate a valid set of program rules by using appropriate (non most general) unifiers before reducing a term. In the example, it suffices with replacing the occurrences of variable `X` by term `s(X)` in each rule (as our needed narrowing based unfolding rule does), which is the key point to restore the required structure of the transformed program.

4 The Tupling Strategy

The tupling strategy was introduced in [10, 13] to optimize functional programs. This strategy is very effective when several functions require the computation of the same subexpression, in which case we tuple together those functions. By avoiding either multiple accesses to data structures or common subcomputations one often gets linear recursive programs (i.e., programs whose rhs's have at most one recursive call) from nonlinear recursive programs [21].

This kind of optimization has not been fully achieved by using automatic transformation techniques such as composition or partial evaluation [4]. Since tupling has not been automated in general in the specialized literature, our proposal consists of decomposing it in three stages and try to automate each one of them in order to generate a fully automatic tupling algorithm. Each stage may consists of several steps done with the transformation rules presented in section 3. We focus our attention separately in the following transformation stages: definition introduction, unfolding and abstraction+folding phase.

4.1 Definition Introduction Phase

This phase corresponds to the so-called eureka generation phase, which is the key point for a transformation strategy to proceed[3]. The problem of achieving an appropriate set of eureka definitions is well-known in the literature related to fold/unfold transformations [10, 20, 22, 4]. For the case of the composition strategy, eureka definitions can be easily identified since they correspond to nested calls, although more refined static analysis (like the one presented in [4], which is based on partial evaluation) may obtain optimal sets of eureka definitions in one go.

On the other hand, the problem of finding appropriate eureka definitions for the tupling strategy is much more difficult, mainly due to the fact that the

[3] Sometimes it is directly called "tupling" [20], but here we prefer to reserve this word for the global algorithm.

calls to be tupled are not nested and they may be arbitrarily distributed in the right hand side of a rule. Sophisticated static analysis have been developed in the literature using dependencies graphs ([11, 19]), m-dags ([8]), symbolic trace trees [9] and other intrincated structures. The main problems appearing in such approaches are that the analysis are not as general as wanted (they can fail even although the program admits tupling optimizations), they are time and space consuming and/or they may duplicate some work too[4].

Table 1. Definition_Introduction_Phase

INPUT:	Original Program $\mathcal{R}$ Program Rule $R = (l \to r)$
BODY:	1. Let $T = \langle t_1, \ldots, t_n \rangle$ $(n > 1)$ be a tuple where $\{t_1, \ldots, t_n\}$ is the set of operation-rooted subterms of r that are *innermost* (i.e., t_i does not contain operation-rooted subterms) such that each one of them shares at least a common variable with at least one more subterm in T 2. Apply the DEFINITION INTRODUCTION RULE to generate : $R_{def} = (f_{new}(\overline{x}) \to T)$ where f_{new} is a new function symbol not appearing $\mathcal{R}$, and $\overline{x}$ is the set of variables of T
OUTPUT:	Definition Rule (Eureka) R_{def}

In order to avoid these risks, our approach generates eureka definition following a very simple strategy (Table 1) that obtains similar levels of generality than previous approaches and covers most practical cases. The main advantages are that the analysis is terminating, easy and quickly, and does not perform redundant calculus (like unfolding steps, instantiations, etc.) that properly corresponds to subsequent phases.

As illustrated by step 1 in Example 2, our eureka generator proceeds as the algorithm in Table 1 shows. Observe that the input of the algorithm is the original program $\mathcal{R}$ and a selected rule $R \in \mathcal{R}$ which definition is intended to be

[4] This fact is observed during the so-called "program extraction phase" in [20]. This kind of post-processing can be made directly (which requires to store in memory elaborated data structures, as is the case of symbolic computation trees) or via transformation rules similarly to our method, but with the disadvantage in that case that many folding/unfolding steps done at "eureka generation time" must be redundantly redone afterwards at "transformation time" or during the program extraction phase.

optimized by tupling. In the worst case, every rule in the program could be used as input, but only those that generate appropriate eureka definitions should be considered afterwards in the global tupling algorithm.

One final remark: it is not clear in general neither the number nor the occurrences of calls to be tupled, but some intuitions exist. Similarly to most classical approaches in the literature, we require that only terms sharing common variables be tupled in the rhs of the eureka definition [11, 20]. On the other hand, since it is not usual that terms to be tupled contain operation rooted terms as parameters, we cope with this fact in our definition by requiring that only operation-rooted subterms[5] at innermost positions of r be collected. On the contrary, the considered subterms would contain nested calls which should be more appropriately transformed by composition instead of tupling (see [4] for details).

4.2 Unfolding Phase

During this phase, that corresponds to the so-called *symbolic computation* in many approaches (see a representative list in [20]), the eureka definition $R_{def} = (f_{new}(\overline{x}) \to T)$ generated in the previous phase, is unfolded possibly several times (at least once) using the original program $\mathcal{R}$, and returning a new program $\mathcal{R}_{unf}$, which represents the unfolded definitions of f_{new} . Since the rhs of the rule to be transformed is a tuple of terms (T), the subterm to be unfolded can be decided in a "don't care" non-deterministic way. In our case, we follow a criterium that is rather usual in the literature ([10, 11]), and we give priority to such subterms where recursion parameters are less general than others, as occurs in step 2 of Example 2 (where we unfold the term `fib(s(X))` instead of `fib(X)`). Moreover, we impose a new condition: each unfolding step must be followed by normalizing steps as much as possible, as illustrates step 3 in Example 2.
As shown in Table 2, the unfolding phase basically consists of a repeated application of the unfolding rule defined in Section 3. This is done by calling function `UNFOLD` with the set of rules that must be eventually unfolded. In each iteration, once a rule in $\mathcal{R}_{unf}$ is unfolded, it is removed and replaced with the rules obtained after unfolding it in the resulting program $\mathcal{R}_{unf}$, which is dynamically actualized in our algorithm. Initially, program $\mathcal{R}_{unf}$ only contains the definition rule $R_{def} = (f_{new}(\overline{x}) \to T)$. We observe that, once the process has been started, any rule obtained by application of the unfolding rule has the form $R' = (\theta(f_{new}(\overline{x})) \to T')$. In order to stop the process, we must check if each rule $R' \in \mathcal{R}_{unf}$ verifies one of the following conditions:

- **Stopping condition 1:** If there are no subterms in T' sharing common variables[6] or they can not be narrowed anymore, then rule R' represents a

[5] In order to relax this conditions, we let the user to remove the rules defining a given symbol f before the transformation process begins. Afther this, symbol f will be considered similarly to any other primitive or constructor symbol in the program.
[6] This novel stopping condition produces good terminating results during the search for regularities since it is more relaxed than other ones considered in the literature.

Table 2. Unfolding_Phase

`INPUT:`	`Original Program` $\mathcal{R}$ `Definition Rule (Eureka)` R_{def}
`BODY:`	1. `Let` $\mathcal{R}_{unf} = \{R_{def}\}$ `be a program` 2. `Repeat` $\mathcal{R}_{unf}$`=UNFOLD(`$\mathcal{R}_{unf}, \mathcal{R}$`)` `until every rule` $R' \in \mathcal{R}_{unf}$ `verifies TEST(`R', R_{def}`)>0`
`OUTPUT:`	`Unfolded Program` $\mathcal{R}_{unf}$

case base definition for the new symbol f_{new}. This fact is illustrated by step 3 and rule R_7 in Example 2.

- **Stopping condition 2:** There exists a substitution θ and a tuple T'' that packs the set of different innermost operation-rooted subterms that share common variables in T' (without counting repetitions), such that $\theta(T) = T''$[7]. Observe that rule R_6 verifies this condition in Example 2 since, for $\theta = id$ there exists (in its rhs) two an one instances respectively of the terms ocurring in the original tuple, that is, $\theta(T) = T = T'' = \langle \mathtt{f(s(X))}, \mathtt{f(X)} \rangle$.

In algorithm of Table 2 these terminating conditions are checked by function `TEST`, which obviously requires the rules whose rhs's are the intended tuples T and T'. Codes 0, 1 and 2 are returned by `TEST` when none, the first or the second stopping condition hold, respectively. We assume that, when `TEST` returns code 0 forever, the `repeat` loop is eventually aborted and then, the unfolding process returns an empty program (that will abort the whole tupling process too).

4.3 Abstraction+Folding Phase

This phase follows the algorithm shown in Table 3 and is used twice, not only for obtaining efficient recursive definitions of the new symbol f_{new} (defined initially by the eureka rule $\mathcal{R}_{def}$ which is always used as second argument), but also for redefine old function symbols in terms of the optimized definition of f_{new}. This fact depends on the first parameter used when calling the algorithm, which may be the unfolded program obtained in the previous phase ($\mathcal{R}_{unf}$), or the original program ($\mathcal{R}$), respectively. In both cases, the algorithm acts in the same

[7] This condition is related to the so-called *similarity*, *regularity* or *foldability* conditions in other approaches ([20]) since it suffices to enable subsequent abstraction+folding steps which may lead to efficient recursive definitions of f_{new}.

way returning the abstracted and folded program $\mathcal{R}'$[8]. We consider both cases separately.

Table 3. Abstraction+Folding_Phase

INPUT:	`Program` $\mathcal{R}_{aux}$ `such that` $\mathcal{R}_{aux} = \mathcal{R}_{unf}$ `or` $\mathcal{R}_{aux} = \mathcal{R}$ `Definition Rule (Eureka)` R_{def}
BODY:	1. `Let` $\mathcal{R}' = \mathcal{R}_{aux}$ `be a program and let` R' `be an empty rule` 2. `For every rule` $R \in \mathcal{R}_{aux}$ `verifying TEST(`R, R_{def}`)=2` R'`=ABSTRACT(`R, R_{def}`)` R'`=FOLD(`R', R_{def}`)` $\mathcal{R}'$=$\mathcal{R}' - \{R\} \cup \{R'\}$
OUTPUT:	`Abstracted+Folded Program` $\mathcal{R}'$

Remember that $R_{def} = (f_{new}(\overline{x}) \to T)$ where $T = \langle t_1, \ldots, t_n \rangle$. As said before, for every rule $R = (\sigma(f_{new}(\overline{x})) \to r') \in \mathcal{R}_{unf}$ satisfying `TEST(`R, R_{def}`)=2` there exists sequences of disjoint positions[9] $\overline{P_j}$ in $\mathcal{FP}os(r')$ where $r'|_p = \theta(t_j)$ for all p in P_j, $j = 1, \ldots, n$, i.e., $r' = r'[\theta(\overline{t_j})]_{\overline{P_j}}$. Hence, it is possible to apply the abstraction rule described in Section 3 to R. This step is done by function `ABSTRACT(`R, R_{def}`)`, that abstract R accordingly to tuple T in rule R_{def} and generates the new rule R':

$$\sigma(f_{new}(\overline{x})) \;\to\; r'[\overline{z_j}]_{\overline{P_j}} \;\; where\; \langle z_1, \ldots, z_n \rangle = \theta(\langle t_1, \ldots, t_n \rangle)\ .$$

where $\overline{z_j}$ are fresh variables. This fact is illustrated by step 4 and rule R_8 in Example 2.

Observe now that this last rule can be folded by using the eureka definition R_{def}, since all the applicability conditions required by our folding rule (see Definition 1) are fulfilled. This fact is done by the call to `FOLD(`R', R_{def}`)` which returns the new rule:

$$\sigma(f_{new}(\overline{x})) \;\to\; r'[\overline{z_j}]_{\overline{P_j}} \;\; where\; \langle z_1, \ldots, z_n \rangle = \theta(f_{new}(\overline{x})).$$

Note that this rule (illustrated by rule R_9 and step 5 in Example 2), as any other rule generated (and accumulated in the resulting program $\mathcal{R}'$) in this phase, corresponds to a recursive definition of f_{new}, as desired.

[8] This program will be afterwards called $\mathcal{R}_{fold}$ or $\mathcal{R}_{post}$ depending whether $\mathcal{R}_{aux}$ be $\mathcal{R}_{unf}$ or $\mathcal{R}$, respectively.

[9] This positions obviously correspond to the set of innermost operation-rooted subterms that share common variables in r'.

Let us now consider the case in which $\mathcal{R}_{aux}$ is the original program $\mathcal{R}$. In contrast with the previous case, where an optimal definition of f_{new} is obtained, now the process tries with rules in $\mathcal{R}$ in order to replace and reuse as much as possible the optimized definition of f_{new} into the original program $\mathcal{R}$. In particular, we know that there exists at least a rule $R = (l \to r) \in \mathcal{R}$ (the one considered in Table 1 to generate de eureka R_{def}) that verifies $\texttt{TEST}(R, R_{def})\texttt{=2}$. Hence, similarly as done before, we can apply an abstraction and folding steps to it, obtaining the new rule:

$$l \;\to\; r[\overline{z_j}]_{\overline{P_j}} \;\; where \; \langle z_1, \ldots, z_n \rangle = f_{new}(\overline{x}).$$

where the call to f_{new} enhances the final definition of the old function symbol that roots l. Steps 6 and 7, and rules R_{10} and R_{11} in Example 2 illustrate these last steps of the abstraction+folding phase.

The following example shows in detail how to optimize by tupling a program by simple chaining the three phases described before.

Example 4. The average of the elements of a list can be computed by the following program $\mathcal{R}_0$:

(R_1) $\quad \texttt{av(L)} \to \texttt{sum(L)/len(L)}$
(R_2) $\quad \texttt{sum(nil)} \to \texttt{0}$
(R_3) $\quad \texttt{sum(X : T)} \to \texttt{X} + \texttt{sum(T)}$
(R_4) $\quad \texttt{len(nil)} \to \texttt{0}$
(R_5) $\quad \texttt{len(X : T)} \to \texttt{1} + \texttt{len(T)}$

(together with the rules for the primitive symbols / and +). Observe that this program traverses a given list twice, which can be reduced to once by applying the tupling strategy as follows:

1. **Definition Introduction Phase**
   (R_6) $\quad \texttt{new(L)} \to \langle \texttt{sum(L)}, \texttt{len(L)} \rangle$
2. **Unfolding Phase**
   (a) Unfolding of rule R_6, (narrowing subterm $\texttt{sum(L)}$):
      (R_7) $\quad \texttt{new(nil)} \to \langle \texttt{0}, \texttt{len(nil)} \rangle$
      (R_8) $\quad \texttt{new(X : T)} \to \langle \texttt{X} + \texttt{sum(T)}, \texttt{len(X : T)} \rangle$
   (b) Normalizing-unfolding steps on R_7 and R_8:
      (R_9) $\quad \texttt{new(nil)} \to \langle \texttt{0}, \texttt{0} \rangle$
      (R_{10}) $\quad \texttt{new(X : T)} \to \langle \texttt{X} + \texttt{sum(T)}, \texttt{1} + \texttt{len(T)} \rangle$
3. **Abstraction+Folding Phase**
   (a) Abstraction of R_{10}:
      (R_{11}) $\quad \texttt{new(X : T)} \to \langle \texttt{X} + \texttt{Z}_1, \texttt{1} + \texttt{Z}_2 \rangle \; \texttt{where} \; \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \langle \texttt{sum(T)}, \texttt{len(T)} \rangle$
   (b) Folding of $\langle \texttt{sum(T)}, \texttt{len(T)} \rangle$ in rule R_8 using R_6:
      (R_{12}) $\quad \texttt{new(X : T)} \to \langle \texttt{X} + \texttt{Z}_1, \texttt{1} + \texttt{Z}_2 \rangle \; \texttt{where} \; \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \texttt{new(T)}$
   (c) Abstraction of R_1:
      (R_{13}) $\quad \texttt{av(L)} \to \texttt{Z}_1/\texttt{Z}_2 \; \texttt{where} \; \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \langle \texttt{sum(L)}, \texttt{len(L)} \rangle$
   (d) Folding of $\langle \texttt{sum(L)}, \texttt{len(L)} \rangle$ in rule R_{13} using again rule R_6:
      (R_{14}) $\quad \texttt{av(L)} \to \texttt{Z}_1/\texttt{Z}_2 \; \texttt{where} \; \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \texttt{new(L)}$

Now, the (enhanced) transformed program is:

$$
\begin{array}{ll}
(R_{14}) & \quad \mathtt{av(L)} \to \mathtt{Z_1/Z_2 \ where} \ \langle \mathtt{Z_1, Z_2} \rangle = \mathtt{new(L)} \\
(R_9) & \mathtt{new(nil)} \to \langle \mathtt{0, 0} \rangle \\
(R_{12}) & \mathtt{new(X : T)} \to \langle \mathtt{X + Z_1, 1 + Z_2} \rangle \ \mathtt{where} \ \langle \mathtt{Z_1, Z_2} \rangle = \mathtt{new(T)}
\end{array}
$$

5 The Automatic Tupling Algorithm

In this section, we use the three transformation phases presented in the previous section to define a realistic, incremental and automatic tupling algorithm that optimizes most practical cases. Firstly, we propose and example that shows the limitations of using our eureka generator in a naive way.

Example 5. Consider the classical problem of Hanoi's Towers, that can be solved in a natural but inefficient way by using the following program $\mathcal{R}_0$:

$$
\begin{array}{rl}
\mathtt{h(0, A, B, C)} \to \mathtt{nil} & (R_1) \\
\mathtt{h(s(N), A, B, C)} \to \mathtt{app(h(N, A, C, B), mov(A, B) : h(N, C, B, A))} & (R_2)
\end{array}
$$

(together with rules defining the typical function that concatenates two lists `app`). Similarly to Example 2, this program has an exponential complexity due to the two recursive calls performed in the rhs of rule R_2, and it also admits a tupling based optimization that reduces its complexity to linear. As showed in [11], the eureka tuple needed to solve this problem (in a pure functional setting) is: $\mathtt{T} = \langle \mathtt{h(N, A, C, B), h(N, C, B, A), h(N, B, A, C)} \rangle$, and observe that such tuple is composed by three calls instead of two ones as our eureka generator would produce. On the other hand, we observe that an abstraction operation performed on rule R_2 for packing the three calls of `T`, is not available in our functional-logic setting, since our abstraction rule may introduce forbidden extra-variables in the abstracted rule (it tries to link to a new free variable a subexpression -$\mathtt{h(N, B, A, C)}$- not existing in the original rule!).

At this point, we have found a contradiction: if we try to perform tupling with the intended eureka (three calls) we need to redefine not only our eureka generator but also our abstraction rule[10] but if we use the eureka built by our eureka

[10] As stated in [3], our abstraction rule is internally defined in terms of definition introduction and folding steps (which ensures the correctness of the transformation), and simulates a kind of lambda lifting that generates two rules without using the `where` construct. Roughly speaking, after abstracting rule R_2 in example 5 we obtain something similar to:

$$
\begin{array}{rl}
\mathtt{h(s(N), A, B, C)} \to \mathtt{new(A, B, h(N, A, C, B), h(N, C, B, A), \underline{h(N, B, A, C)})} & (R') \\
\mathtt{new(A, B, Z_1, Z_2, \underline{Z_3})} \to \mathtt{app(Z_1, mov(A, B) : Z_2)} & (R'')
\end{array}
$$

Our method implements this (lambda-lifting based) abstraction step by considering that those rules have been obtained by a definition introduction step (that generates rule R'') followed by a folding step (generating rule R' by folding rule R_2 using rule R''). And now observe that rule R'' is not a valid eureka definition (variable Z_3 is not used in the rhs of rule R''), which prevents -in general- the abstraction of expressions (i.e., $\mathtt{h(N, B, A, C)}$) not appearing in the rhs of the rule to be abstracted.

generator (two calls) the transformation process may not culminate. So, which is the intended eureka we must use in this case?. Fortunately there exists a positive answer to this question: use both eurekas!. This process can be done in an incremental way[11], that is, firstly we generate and use the eureka with two calls, and eventually we hope that the second eureka (with three calls) be generated and used. Let us continue with the example.

Example 6. The definition introduction and unfolding phases (as described in Section 4) for our example are:

1. Definition Introduction Phase:

   $\mathtt{new(N,A,B,C)} \to \langle \mathtt{h(N,A,B,C),h(N,B,C,A)} \rangle$ (R_3)

2. Unfolding Phase:

   (a) Unfolding of rule R_3 narrowing redex $\mathtt{h(N,A,B,C)}$:

   $$\begin{aligned} \mathtt{new(0,A,B,C)} &\to \langle \mathtt{nil,h(0,B,C,A)} \rangle & (R_4) \\ \mathtt{new(s(N),A,B,C)} &\to \langle \mathtt{app(h(N,A,C,B),mov(A,B):h(N,C,B,A)),} \\ &\quad\ \mathtt{h(s(N),B,C,A)} \rangle & (R_5) \end{aligned}$$

   (b) Unfold (normalize) rule R_4 reducing $\mathtt{h(0,B,C,A)}$ (Base Case Definition):

   $\mathtt{new(0,A,B,C)} \to \langle \mathtt{nil,nil} \rangle$ (R_6)

   (c) Unfolding (normalizing) of rule R_5 narrowing redex $\mathtt{h(s(N),B,C,A)}$:

   $$\begin{aligned} \mathtt{new(s(N),A,B,C)} &\to \langle \mathtt{app(h(N,A,C,B),mov(A,B):h(N,C,B,A)),} \\ &\quad\ \mathtt{app(h(N,B,A,C),mov(B,C):h(N,A,C,B))} \rangle \ (R_7) \end{aligned}$$

At this point we consider that the unfolding phase is finished since TEST(R_6, R_3)=1 and TEST(R_7, R_3)=2. However, observe that applying abstraction+folding steps to R_7 will not generate an optimized definition, since the rhs of R_7 contains four calls to h, where three of them are differents. So, after abstraction+folding steps, the recursive definition of new would have two calls: one for new and one for h, which does not represent an enhanced definition.

On the other hand, we think that it would be more appropriate to regenerate a new tupling process in order to optimize R_7, since now our eureka generator is able to produce an eureka with three calls, as desired.

Example 7. The new tupling process consider as initial program:

$$\begin{aligned} \mathtt{h(0,A,B,C)} &\to \mathtt{nil} & (R_1) \\ \mathtt{h(s(N),A,B,C)} &\to \mathtt{app(h(N,A,C,B),mov(A,B):h(N,C,B,A))} & (R_2) \\ \mathtt{new(0,A,B,C)} &\to \langle \mathtt{nil,nil} \rangle & (R_6) \\ \mathtt{new(s(N),A,B,C)} &\to \langle \mathtt{app(h(N,A,C,B),mov(A,B):h(N,C,B,A)),} \\ &\quad\ \mathtt{app(h(N,B,A,C),mov(B,C):h(N,A,C,B))} \rangle & (R_7) \end{aligned}$$

And now we want to optimize rule R_7 as follows:

[11] This intuitive idea is also present in [20], but we have more precisely identified the required conditions that automate the process.

1. Definition Introduction Phase:

   $\texttt{new2(N,A,B,C)} \to \langle \texttt{h(N,A,B,C)}, \texttt{h(N,B,C,A)}, \texttt{h(N,C,A,B)} \rangle$ (R_8)

2. Unfolding Phase:

   (a) Unfolding of rule R_8 narrowing redex $\texttt{h(N,A,B,C)}$:

   $$\begin{array}{rll} \texttt{new2(0,A,B,C)} & \to \langle \texttt{nil}, \texttt{h(0,B,C,A)}, \texttt{h(0,C,A,B)} \rangle & (R_9) \\ \texttt{new2(s(N),A,B,C)} & \to \langle \texttt{app(h(N,A,C,B),mov(A,B)} : \texttt{h(N,C,B,A))}, & \\ & \quad \texttt{h(s(N),B,C,A)}, \texttt{h(s(N),C,A,B)} \rangle & (R_{10}) \end{array}$$

   (b) Unfold (normalize) rule R_9 reducing $\texttt{h(0,B,C,A)}$:

   $\texttt{new2(0,A,B,C)} \to \langle \texttt{nil}, \texttt{nil}, \texttt{h(0,C,A,B)} \rangle$ (R_{11})

   (c) Unfolding (normalizing) of rule R_{11} exploiting redex $\texttt{h(0,C,A,B)}$:

   $\texttt{new2(0,A,B,C)} \to \langle \texttt{nil}, \texttt{nil}, \texttt{nil} \rangle$ (R_{12})
   Observe now that TEST(R_{12}, R_8)=1 since R_{12} represents a base case definition for new2.

   (d) Unfold (normalize) rule R_{10} narrowing redex $\texttt{h(s(N),B,C,A)}$:

   $$\begin{array}{rll} \texttt{new2(s(N),A,B,C)} & \to \langle \texttt{app(h(N,A,C,B),mov(A,B)} : \texttt{h(N,C,B,A))}, & \\ & \quad \texttt{app(h(N,B,A,C),mov(B,C)} : \texttt{h(N,A,C,B))}, & \\ & \quad \texttt{h(s(N),C,A,B)} \rangle & (R_{13}) \end{array}$$

   (e) And finally, unfolding (normalizing) of rule R_{13} reducing $\texttt{h(s(N),C,A,B)}$:

   $$\begin{array}{rl} \texttt{new2(s(N),A,B,C)} & \to \langle \texttt{app(h(N,A,C,B),mov(A,B)} : \texttt{h(N,C,B,A))}, \\ & \quad \texttt{app(h(N,B,A,C),mov(B,C)} : \texttt{h(N,A,C,B))}, \\ & \quad \texttt{app(h(N,C,B,A),mov(C,A)} : \texttt{h(N,B,A,C))} \rangle \; (R_{14}) \end{array}$$

   Now observe that the unfolding process ends since TEST(R_{14}, R_8)=2, and $R_{unf} = \{R_{12}, R_{14}\}$.

3. Abstraction+Folding Phase:

   Firstly we act on $\mathcal{R}_{unf}$ in order to generate the recursive definition for new2.

   (a) Abstraction of rule R_{14} (using R_8):

   $$\begin{array}{rcl} \texttt{new2(s(N),A,B,C)} & \to & \langle \texttt{app(Z}_1\texttt{,mov(A,B)} : \texttt{Z}_2\texttt{)}, \\ & & \;\texttt{app(Z}_3\texttt{,mov(B,C)} : \texttt{Z}_1\texttt{)}, \\ & & \;\texttt{app(Z}_2\texttt{,mov(C,A)} : \texttt{Z}_3\texttt{)} \rangle \\ \texttt{where } \langle \texttt{Z}_1, \texttt{Z}_2, \texttt{Z}_3 \rangle & = & \langle \texttt{h(N,A,C,B)}, \texttt{h(N,C,B,A)}, \texttt{h(N,B,A,C)} \rangle \; (R_{15}) \end{array}$$

   (b) Folding of $\langle \texttt{h(N,A,C,B)}, \texttt{h(N,C,B,A)}, \texttt{h(N,B,A,C)} \rangle$ in rule R_{15} using rule R_8:

   $$\begin{array}{rl} \texttt{new2(s(N),A,B,C)} & \to \langle \texttt{app(Z}_1\texttt{,mov(A,B)} : \texttt{Z}_2\texttt{)}, \\ & \quad \texttt{app(Z}_3\texttt{,mov(B,C)} : \texttt{Z}_1\texttt{)}, \\ & \quad \texttt{app(Z}_2\texttt{,mov(C,A)} : \texttt{Z}_3\texttt{)} \rangle \\ & \quad \texttt{where } \langle \texttt{Z}_1, \texttt{Z}_2, \texttt{Z}_3 \rangle = \texttt{new2(N,A,C,B)} \; (R_{16}) \end{array}$$

   Secondly, we try to redefine new in terms of new2.

   (a*) Abstraction of rule R_7 (using R_8):

$$\begin{array}{rcl} \mathtt{new(s(N),A,B,C)} & \rightarrow & \langle \mathtt{app(Z_1,mov(A,B):Z_2),app(Z_3,mov(B,C):Z_1)} \rangle \\ \mathtt{where}\ \langle \mathtt{Z_1,Z_2,Z_3} \rangle & = & \langle \mathtt{h(N,A,C,B),h(N,C,B,A),h(N,B,A,C)} \rangle \quad (R_{17}) \end{array}$$

(b*) Folding $\langle \mathtt{h(N,A,C,B),h(N,C,B,A),h(N,B,A,C)} \rangle$ in rule R_{17} using rule R_8:

$$\begin{array}{l} \mathtt{new(s(N),A,B,C)} \rightarrow \langle \mathtt{app(Z_1,mov(A,B):Z_2),app(Z_3,mov(B,C):Z_1)} \rangle \\ \qquad\qquad \mathtt{where}\ \langle \mathtt{Z_1,Z_2,Z_3} \rangle = \mathtt{new2(N,A,C,B)} \quad (R_{18}) \end{array}$$

Thanks to this re-application of the whole tupling algorithm, now we have an optimal definition not only for `new2`, but also for `new`. Then, we can recover the first tupling loop at the point we left it in Example 6 in order to reuse the optimal definition generated in Example 7 (second tupling iteration) as follows.

Example 8. We now perform the pending Abstraction+Folding phase of Example 6. Since R_{unf} has been optimized in Example 7, no more abstraction and folding steps are needed on it. So, this phase only acts on rules belonging to the original program $\mathcal{R}_0$ in order to reuse the enhanced definition of `new`.

1. Abstraction of rule R_2 (using R_3):

$$\begin{array}{l} \mathtt{h(s(N),A,B,C)} \rightarrow \mathtt{app(Z_1,mov(A,B):Z_2)} \\ \qquad\qquad \mathtt{where}\ \langle \mathtt{Z_1,Z_2} \rangle = \langle \mathtt{h(N,A,C,B),h(N,C,B,A)} \rangle\ (R_{19}) \end{array}$$

2. Folding of $\langle \mathtt{h(N,A,C,B),h(N,C,B,A))} \rangle$ in rule R_{19} using the eureka R_3:

$$\mathtt{h(s(N),A,B,C)} \rightarrow \mathtt{app(Z_1,mov(A,B):Z_2)}\ \mathtt{where}\ \langle \mathtt{Z_1,Z_2} \rangle = \mathtt{new(N,A,C,B)}\ (R_{20})$$

Now, the final program (with rules $R_1, R_{20}, R_6, R_{18}, R_{12}$ and R_{16}) has a linear complexity thanks to the use of enhanced definitions for the new symbols `new` and `new2` which are created in an incremental way without producing redundant calculi.

$$\begin{array}{rcl} \mathtt{h(0,A,B,C)} & \rightarrow & \mathtt{nil} \\ \mathtt{h(s(N),A,B,C)} & \rightarrow & \mathtt{app(Z_1,mov(A,B):Z_2)}\ \mathtt{where}\ \langle \mathtt{Z_1,Z_2} \rangle = \mathtt{new(N,A,C,B)} \\ \mathtt{new(0,A,B,C)} & \rightarrow & \langle \mathtt{nil,nil} \rangle \\ \mathtt{new(s(N),A,B,C)} & \rightarrow & \langle \mathtt{app(Z_1,mov(A,B):Z_2),app(Z_3,mov(B,C):Z_1)} \rangle \\ & & \mathtt{where}\ \langle \mathtt{Z_1,Z_2,Z_3} \rangle = \mathtt{new2(N,A,C,B)} \\ \mathtt{new2(0,A,B,C)} & \rightarrow & \langle \mathtt{nil,nil,nil} \rangle \\ \mathtt{new2(s(N),A,B,C)} & \rightarrow & \langle \mathtt{app(Z_1,mov(A,B):Z_2),\ app(Z_3,mov(B,C):Z_1)}, \\ & & \mathtt{app(Z_2,mov(C,A):Z_3)} \rangle \mathtt{where} \langle \mathtt{Z_1,Z_2,Z_3} \rangle = \mathtt{new2(N,A,C,B)} \end{array}$$

All these facts have been taken into account in our final version of the automatic tupling algorithm which is presented in Table 4. The call inside the algorithm that decides (during the unfolding phase) when applying a new tupling iteration is represented by `TEST` that returns code 3 when the following new stopping condition holds (we assume that the eureka definition is $R_{def} = (f_{new}(\overline{x}) \rightarrow \langle t_1, \ldots, t_n \rangle)$ and the unfolded rule $R' = (\theta(f_{new}(\overline{x})) \rightarrow T')$):

– **Stopping condition 3:** Stopping conditions 1 and 2 do not hold and the number m of innermost operation rooted subterms (including the repeated ones) that share common variables in T' is greater than n (i.e., $m > n$)[12].

[12] This situation -illustrated by rule R_7 in example 7- reveals that better optimizations can be achieved by packing more calls in a tuple.

Finally, we assume that the incremental process may be aborted if it overflows a maximal number of allowed tupling recursive calls.

Table 4. Tupling_Algorithm

INPUT:	Original Program $\mathcal{R}$ Program Rule R
BODY:	1. Let R_{def} be an empty rule 2. Let $\mathcal{R}_{unf}, \mathcal{R}_{fold}$, $\mathcal{R}_{post}$ and $\mathcal{R}'$ be empty programs 3. R_{def}=Definition_Introduction_Phase($\mathcal{R}, R$) 4. $\mathcal{R}_{unf}$=Unfolding_Phase($\mathcal{R}, R_{def}$) 5. For every rule $R' \in \mathcal{R}_{unf}$ verifying TEST(R', R_{def})=3 $\mathcal{R}_{unf}$=Tupling_Algorithm($\mathcal{R} \cup \mathcal{R}_{unf}, R'$) 6. $\mathcal{R}_{fold}$=Abstraction+Folding_Phase($\mathcal{R}_{unf}, R_{def}$) 7. $\mathcal{R}_{post}$=Abstraction+Folding_Phase($\mathcal{R}, R_{def}$) 8. $\mathcal{R}' = \mathcal{R}_{fold} \cup \mathcal{R}_{post}$
OUTPUT:	Transformed Program $\mathcal{R}'$

Example 9. The following example is recasted from [11], where it is proposed as a difficult case for obtaining the appropriate eureka definitions. Following our method, the eureka definitions are produced in an incremental way, which seems easier than the more involved analysis presented in [11]. The program calculates a list of factorials (for even numbers) as follows.

$$
\begin{array}{lll}
\mathtt{fl(0)} & \to \mathtt{nil} & (R_1) \\
\mathtt{fl(s(0))} & \to \mathtt{nil} & (R_2) \\
\mathtt{fl(s(s(X)))} & \to \mathtt{f(s(s(X))):fl(X)} & (R_3) \\
\mathtt{f(0)} & \to \mathtt{s(0)} & (R_4) \\
\mathtt{f(s(X))} & \to \mathtt{s(X)} * \mathtt{f(X)} & (R_5)
\end{array}
$$

Together with the rules for $*$, which is considered (and treated similarly to a constructor symbol) a primitive symbol in our setting. The transformation process proceeds as follows.

1. Definition Introduction Phase:

$$\mathtt{new(X)} \to \langle \mathtt{f(s(s(X)))}, \mathtt{fl(X)} \rangle \qquad (R_6)$$

2. Unfolding Phase:

(a) Unfolding (normalizing) of rule R_6, reducing subterm $\mathtt{f(s(s(X)))}$:
$$\mathtt{new(X)} \to \langle \mathtt{s(s(X))} * \mathtt{f(s(X))}, \mathtt{fl(X)} \rangle \qquad (R_7)$$
(b) Unfolding (normalizing) of rule R_7, reducing subterm $\mathtt{f(s(X))}$
$$\mathtt{new(X)} \to \langle \mathtt{s(s(X))} * \mathtt{s(X)} * \mathtt{f(X)}, \mathtt{fl(X)} \rangle \qquad (R_8)$$
(c) Unfolding of rule R_8, reducing subterm $\mathtt{f(X)}$
$$\mathtt{new(0)} \to \langle \mathtt{s(s(0))} * \mathtt{s(0)} * \mathtt{s(0)}, \mathtt{fl(0)} \rangle \qquad (R_9)$$
$$\mathtt{new(s(X))} \to \langle \mathtt{s(s(s(X)))} * \mathtt{s(s(X))} * \mathtt{s(X)} * \mathtt{f(X)}, \mathtt{fl(s(X))} \rangle \qquad (R_{10})$$
(d) Normalizing of rule R_9, reducing subterm $\mathtt{fl(0)}$
$$\mathtt{new(0)} \to \langle \mathtt{s(s(0))} * \mathtt{s(0)} * \mathtt{s(0)}, \mathtt{nil} \rangle \qquad (R_{11})$$
(e) Unfolding of rule R_{10}, reducing subterm $\mathtt{fl(s(X))}$
$$\mathtt{new(s(0))} \to \langle \mathtt{s(s(s(0)))} * \mathtt{s(s(0))} * \mathtt{s(0)} * \mathtt{f(0)}, \mathtt{nil} \rangle \qquad (R_{12})$$
$$\begin{aligned}\mathtt{new(s(s(X)))} \to \langle & \mathtt{s(s(s(s(X))))} * \mathtt{s(s(s(X)))} * \mathtt{s(s(X))} * \mathtt{f(s(X))}, \\ & \mathtt{f(s(s(X))} : \mathtt{fl(X))} \rangle \end{aligned} \qquad (R_{13})$$
(f) Normalizing of rule R_{12}, reducing subterm $\mathtt{f(0)}$
$$\mathtt{new(s(0))} \to \langle \mathtt{s(s(s(0)))} * \mathtt{s(s(0))} * \mathtt{s(0)} * \mathtt{s(0)}, \mathtt{nil} \rangle \qquad (R_{14})$$
(g) Normalizing of rule R_{13}, reducing subterm $\mathtt{f(s(X))}$
$$\begin{aligned}\mathtt{new(s(s(X)))} \to \langle & \mathtt{s(s(s(s(X))))} * \mathtt{s(s(s(X)))} * \mathtt{s(s(X))} * \mathtt{s(X)} * \mathtt{f(X)}, \\ & \mathtt{f(s(s(X)))} : \mathtt{fl(X))} \rangle \end{aligned} \qquad (R_{15})$$
(h) Normalizing of rule R_{15}, reducing subterm $\mathtt{f(s(s(X)))}$
$$\begin{aligned}\mathtt{new(s(s(X)))} \to \langle & \mathtt{s(s(s(s(X))))} * \mathtt{s(s(s(X)))} * \mathtt{s(s(X))} * \mathtt{s(X)} * \mathtt{f(X)}, \\ & \mathtt{s(s(X)} * \mathtt{f(s(X))} : \mathtt{fl(X))} \rangle \end{aligned} \qquad (R_{16})$$
(i) Normalizing of rule R_{16}, reducing subterm $\mathtt{f(s(X))}$
$$\begin{aligned}\mathtt{new(s(s(X)))} \to \langle & \mathtt{s(s(s(s(X))))} * \mathtt{s(s(s(X)))} * \mathtt{s(s(X))} * \mathtt{s(X)} * \mathtt{f(X)}, \\ & \mathtt{s(s(X)} * \mathtt{s(X)} * \mathtt{f(X)} : \mathtt{fl(X))} \rangle \end{aligned} \qquad (R_{17})$$

3. New iteration of tupling[13] starting with rule R_{17}.
   (a) Definition Introduction Phase:
   $$\mathtt{new2(X)} \to \langle \mathtt{f(X)}, \mathtt{fl(X)} \rangle \qquad (R_{18})$$
   (b) Unfolding Phase:
      i. Unfolding of rule R_{18}, reducing subterm $\mathtt{f(X)}$
      $$\mathtt{new2(0)} \to \langle \mathtt{s(0)}, \mathtt{fl(0)} \rangle \qquad (R_{19})$$
      $$\mathtt{new2(s(X))} \to \langle \mathtt{s(X)} * \mathtt{f(X)}, \mathtt{fl(s(X))} \rangle \qquad (R_{20})$$
      ii. Normalizing of rule R_{19}, reducing subterm $\mathtt{fl(0)}$
      $$\mathtt{new2(0)} \to \langle \mathtt{s(0)}, \mathtt{nil} \rangle \qquad (R_{21})$$
      iii. Unfolding of rule R_{20}, reducing subterm $\mathtt{fl(s(X))}$
      $$\mathtt{new2(s(0))} \to \langle \mathtt{s(0)} * \mathtt{f(0)}, \mathtt{nil} \rangle \qquad (R_{22})$$
      $$\begin{aligned}\mathtt{new2(s(s(X)))} \to \langle & \mathtt{s(s(X))} * \mathtt{f(s(X))}, \\ & \mathtt{f(s(s(X))} : \mathtt{fl(X))} \rangle \end{aligned} \qquad (R_{23})$$
      iv. Normalizing of rule R_{22}, reducing subterm $\mathtt{f(0)}$
      $$\mathtt{new2(s(0))} \to \langle \mathtt{s(0)} * \mathtt{s(0)}, \mathtt{nil} \rangle \qquad (R_{24})$$
      v. Normalizing of rule R_{23}, reducing subterm $\mathtt{f(s(X))}$
      $$\begin{aligned}\mathtt{new2(s(s(X)))} \to \langle & \mathtt{s(s(X))} * \mathtt{s(X)} * \mathtt{f(X)}, \\ & \mathtt{f(s(s(X)))} : \mathtt{fl(X))} \rangle \end{aligned} \qquad (R_{25})$$

[13] Observe that at this point function TEST returns code 3, since stopping conditions 1 and 2 do not hold, and there exist three innermost operation-rooted subterms sharing common variables in the rhs of R_{17}: two occurrences of f(X) and one of fl(X).

vi. Normalizing of rule R_{25}, reducing subterm $\mathtt{f(s(s(X)))}$

$$\mathtt{new2(s(s(X)))} \to \langle \mathtt{s(s(X)) * s(X) * f(X)}, \mathtt{s(s(X) * f(s(X)) : fl(X))} \rangle \qquad (R_{26})$$

vii. Normalizing of rule R_{26}, reducing subterm $\mathtt{f(s(X))}$

$$\mathtt{new2(s(s(X)))} \to \langle \mathtt{s(s(X)) * s(X) * f(X)}, \mathtt{s(s(X) * s(X) * f(X) : fl(X))} \rangle \qquad (R_{27})$$

(c) Abstraction+Folding Phase:

i. Abstraction of rule R_{27}

$$\mathtt{new2(s(s(X)))} \to \langle \mathtt{s(s(X)) * s(X) * Z_1}, \mathtt{s(s(X) * s(X) * Z_1 : Z_2)} \rangle \;\; \mathtt{where} \;\; \langle \mathtt{Z_1}, \mathtt{Z_2} \rangle = \langle \mathtt{f(X)}, \mathtt{fl(X)} \rangle \qquad (R_{28})$$

ii. Folding of $\langle \mathtt{f(X)}, \mathtt{fl(X)} \rangle$ in rule R_{28} using the eureka R_{18}

$$\mathtt{new2(s(s(X)))} \to \langle \mathtt{s(s(X)) * s(X) * Z_1}, \mathtt{s(s(X) * s(X) * Z_1 : Z_2)} \rangle \;\; \mathtt{where} \;\; \langle \mathtt{Z_1}, \mathtt{Z_2} \rangle = \mathtt{new2(X)} \qquad (R_{29})$$

iii. Abstraction of rule R_{17}

$$\mathtt{new(s(s(X)))} \to \langle \mathtt{s(s(s(s(X)))) * s(s(s(X))) * s(s(X)) * s(X) * Z_1}, \mathtt{s(s(X) * s(X) * Z_1 : Z_2)} \rangle \;\; \mathtt{where} \;\; \langle \mathtt{Z_1}, \mathtt{Z_2} \rangle = \langle \mathtt{f(X)}, \mathtt{fl(X)} \rangle \qquad (R_{30})$$

iv. Folding of $\langle \mathtt{f(X)}, \mathtt{fl(X)} \rangle$ in rule R_{30} using the eureka R_{18}

$$\mathtt{new(s(s(X)))} \to \langle \mathtt{s(s(s(s(X)))) * s(s(s(X))) * s(s(X)) * s(X) * Z_1}, \mathtt{s(s(X) * s(X) * Z_1 : Z_2)} \rangle \;\; \mathtt{where} \;\; \langle \mathtt{Z_1}, \mathtt{Z_2} \rangle = \mathtt{new2(X)} \qquad (R_{31})$$

4. Abstraction+Folding Phase:

(a) Abstraction of rule R_3

$$\mathtt{fl(s(s(X)))} \to \mathtt{Z_1 : Z_2} \;\; \mathtt{where} \;\; \langle \mathtt{Z_1}, \mathtt{Z_2} \rangle = \langle \mathtt{f(s(s(X)))}, \mathtt{fl(X)} \rangle \qquad (R_{32})$$

(b) Folding of $\langle \mathtt{f(s(s(X)))}, \mathtt{fl(X)} \rangle$ in rule R_{30} using the eureka R_6

$$\mathtt{fl(s(s(X)))} \to \mathtt{Z_1 : Z_2} \;\; \mathtt{where} \;\; \langle \mathtt{Z_1}, \mathtt{Z_2} \rangle = \mathtt{new(X)} \qquad (R_{33})$$

Now, the (enhanced) transformed program is the following:

$$\begin{array}{lrl}
(R_1) & \mathtt{fl(0)} & \to \mathtt{nil} \\
(R_2) & \mathtt{fl(s(0))} & \to \mathtt{nil} \\
(R_{33}) & \mathtt{fl(s(s(X)))} & \to \mathtt{Z_1 : Z_2} \;\; \mathtt{where} \;\; \langle \mathtt{Z_1}, \mathtt{Z_2} \rangle = \mathtt{new(X)} \\
(R_{11}) & \mathtt{new(0)} & \to \langle \mathtt{s(s(0)) * s(0) * s(0)}, \mathtt{nil} \rangle \\
(R_{14}) & \mathtt{new(s(0))} & \to \langle \mathtt{s(s(s(0))) * s(s(0)) * s(0) * s(0)}, \mathtt{nil} \rangle \\
(R_{31}) & \mathtt{new(s(s(X)))} & \to \langle \mathtt{s(s(s(s(X)))) * s(s(s(X))) * s(s(X)) * s(X) * Z_1}, \\
 & & \quad \mathtt{s(s(X) * s(X) * Z_1 : Z_2)} \rangle \\
 & & \quad \mathtt{where} \;\; \langle \mathtt{Z_1}, \mathtt{Z_2} \rangle = \mathtt{new2(X)} \\
(R_{21}) & \mathtt{new2(0)} & \to \langle \mathtt{s(0)}, \mathtt{nil} \rangle \\
(R_{24}) & \mathtt{new2(s(0))} & \to \langle \mathtt{s(0) * s(0)}, \mathtt{nil} \rangle \\
(R_{29}) & \mathtt{new2(s(s(X)))} & \to \langle \mathtt{s(s(X)) * s(X) * Z_1}, \mathtt{s(s(X) * s(X) * Z_1 : Z_2)} \rangle \\
 & & \quad \mathtt{where} \;\; \langle \mathtt{Z_1}, \mathtt{Z_2} \rangle = \mathtt{new2(X)}
\end{array}$$

Observe that the final program admits more improvements by packing some "*" operations into where expressions, but this is out of our scope (during the transformation we simply consider "*" as a constructor symbol). Note also that we have incrementaly generated two eureka definitions to solve the problem. However, if we firstly normalize rule R_3 then we obtain a new rule of the form:

$$(R'_3) \qquad \mathtt{fl(s(s(X)))} \to \mathtt{s(s(X)) * s(X) * f(X) : fl(X)}$$

and now it is possible to optimize the program by only using the second eureka, obtaining as final program:

$$
\begin{array}{lrl}
(R_1) & \mathtt{fl(0)} \to & \mathtt{nil} \\
(R_2) & \mathtt{fl(s(0))} \to & \mathtt{nil} \\
(R_3'') & \mathtt{fl(s(s(X)))} \to & \mathtt{s(s(X))} * \mathtt{s(X)} * \mathtt{Z_1} : \mathtt{Z_2} \;\; \mathtt{where} \;\; \langle \mathtt{Z_1}, \mathtt{Z_2} \rangle = \mathtt{new2(X)}
\end{array}
$$

$$
\begin{array}{lrl}
(R_{21}) & \mathtt{new2(0)} \to & \langle \mathtt{s(0)}, \mathtt{nil} \rangle \\
(R_{24}) & \mathtt{new2(s(0))} \to & \langle \mathtt{s(0)} * \mathtt{s(0)}, \mathtt{nil} \rangle \\
(R_{29}) & \mathtt{new2(s(s(X)))} \to & \langle \mathtt{s(s(X))} * \mathtt{s(X)} * \mathtt{Z_1}, \mathtt{s(s(X)} * \mathtt{s(X)} * \mathtt{Z_1} : \mathtt{Z_2}) \rangle \\
& & \quad \mathtt{where} \;\; \langle \mathtt{Z_1}, \mathtt{Z_2} \rangle = \mathtt{new2(X)}
\end{array}
$$

This final improvement can be easily incorporated to our algorithm by simply requiring that any rule be normalized before being optimized by tupling.

6 Conclusions and Further Research

Tupling is a powerful optimization strategy which can be achieved by fold/unfold transformations and produces better gains of efficciency than other simpler-automatic transformations such composition or PE. This technique eliminates parallel traversals of identical data structures by merging separate (non-nested) loops together into a new recursive function defined on *tuples* (pairs), thus avoiding the traversal of the considered data structures more than once. As it is well-known in the literature, tupling is very complicated and automatic tupling algorithms either result in high runtime cost (which prevents them from being employed in a real system), or they succeed only for a restricted class of programs [11, 12]. Our approach drops out some of these limitations by automating a realistic and practicable tupling algorithm that optimizes functional-logic programs.

There are several important remarks to do regarding our algorithm. A key point is the way in which eureka definitions are derived. In contrast with previous approaches, we show a very simple method that (at a low cost) can derive a good set of eureka definitions in an incremental way without computing redundant calculus. In [3] we proposed an eureka generator based on PE for the composition strategy that exploits the close relationship between both transformations. Unfortunately, the resulting analysis is not applicable for tupling as evidences the fact that the optimization power of PE is strictly lower than tupling, even when considering PE as eureka generator. Moreover, compared with prior approaches in the field of pure functional programming, our method is less involved (we do not require complicate structures for generating appropriate eureka definitions), more efficient (i.e., redundant computations are avoided), elegant (i.e., eureka definitions are produced in an incremental and natural way) and deals with special particularities of the integrated paradigm (i.e., transformed rules are non overlapping and there are no extra variables in their rhs's).

A relevant property of the programs produced by the automatic tupling algorithm is the strong correctness w.r.t. goals containing calls to old defined function symbols, which is inherited by the strong correctness properties of the set of transformation rules presented in Section 3 (see [3] for details). Moreover, since calls to old symbols are redefined and solved by calls to the optimized definitions of new function symbols, our method guaranties gains in efficiency. By using our automatic tupling algorithm, programs with exponential complexities can be sometimes reduced to linear ones, which contrast with other automatic transformation strategies such as composition or partial evaluation, where optimizations never change the algorithmic complexity of a program ([4]).

There is a final remark about the structure of the transformed programs. Looking at their syntax, we distinguish two disjoint set of rules:

- $\mathcal{R}_{fold}$ represents the set of optimized rules defining the new function symbols, where no calls to rules outside from $\mathcal{R}_{fold}$ (appart from those that are not transformed) appear, i.e., the optimized component is "self-contained".
- On the other hand, $\mathcal{R}_{post}$ coincides with the set of rules defining the set of old function symbols. This component is very similar to the original program $\mathcal{R}$, except for certain calls to the old function symbols which have been transformed (by virtue of abstraction and folding steps) into calls to functions of the $\mathcal{R}_{fold}$ optimized component.

This is achieved by considering as eurekas the tuples of calls to be *funded* and optimized in the rhs's of rules in the original program. Finally, we reuse the new enhanced definitions for those eurekas (tuples) as much as possible in the initial program by performing appropriate abstraction and folding steps.

We conclude by mentioning some ideas for further research. During the last three years, we have been developing the prototype transformation system SYNTH [1], which is publicly available at `http://www.info-ab.uclm.es/personal/gmoreno/synthpage.htm`. The system incorporates the basic rules presented so far, includes the fully automatic composition strategy based on [4] and is able to optimize programs written in the language Curry, a modern multi-paradigm declarative language based on needed narrowing which is intended to become a standard in the functional logic community [16, 17]. Nowadays we are incorporating the automatic tupling algorithm presented in this paper to the kernel of the system. From here, we are interesting in opening two new research lines:

1. **To obtain a fully automatic optimizing tool**. This action might culminate by incorporating the global system inside a Curry compiler, where optimizations done at compiling time (in a completely transparent way to the user) might cope with advanced features of the lenguaje (higher order, residuation, etc.). We think that it is possible to obtain additional benefits by performing the transformation process on programs written in a lower level code.
2. **To estimate and monitore the gains of efficiency produced at transformation time**. In this sense, we want to associate a cost/gain label to each local transformation step when building a transformation sequence. This ac-

tion is not only nice for automatic heuristics, but also it seems interesting for to manually guide the application of transformation rules[14].

7 Acknowledgements

My thanks to María Alpuente and Moreno Falaschi for helpful discussions on transformation strategies. I am specially grateful to German Vidal for critical comments and related work that have helped to improve this paper.

References

1. M. Alpuente, M. Falaschi, C. Ferri, G. Moreno, G. Vidal, and I. Ziliotto. The Transformation System SYNTH. Technical Report DSIC-II/16/99, UPV, 1999. Available in URL: `http://www.dsic.upv.es/users/elp/papers.html`.
2. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Safe folding/unfolding with conditional narrowing. In H. Heering M. Hanus and K. Meinke, editors, *Proc. of the International Conference on Algebraic and Logic Programming, ALP'97, Southampton (England)*, pages 1–15. Springer LNCS 1298, 1997.
3. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A Transformation System for Lazy Functional Logic Programs. In A. Middeldorp and T. Sato, editors, *Proc. of the 4th Fuji International Symposyum on Functional and Logic Programming, FLOPS'99, Tsukuba (Japan)*, pages 147–162. Springer LNCS 1722, 1999.
4. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. An Automatic Composition Algorithm for Functional Logic Programs. In V. Hlaváč, K. G. Jeffery, and J. Wiedermann, editors, *Proc. of the 27th Annual Conference on Current Trends in Theory and Practice of Informatics, SOFSEM'2000*, pages 289–297. Springer LNCS 1963, 2000.
5. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, pages 143–157. Springer LNCS 632, 1992.
6. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages, Portland*, pages 268–279, New York, 1994. ACM Press.
7. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
8. R.S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–418, 1980.
9. M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding. *New Generation Computing*, 11(1):47–79, 1992.
10. R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
11. W. Chin. Towards an Automated Tupling Strategy. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, 1993*, pages 119–132. ACM, New York, 1993.

[14] Although nowadays the user can interact with the system in order to decide which transformation rule apply at the next step, transformation sequences generated in such a way are somehow blind and useless in practice.

12. W. Chin, A. Goh, and S. Khoo. Effective Optimisation of Multiple Traversals in Lazy Languages. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA (Technical Report BRICS-NS-99-1)*, pages 119–130. University of Aarhus, DK, 1999.
13. J. Darlington. Program transformation. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, pages 193–215. Cambridge University Press, 1982.
14. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam, 1990.
15. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
16. M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.
17. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available in `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1999.
18. G. Huet and J.J. Lévy. Computations in orthogonal rewriting systems, Part I + II. In J.L. Lassez and G.D. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–443. The MIT Press, Cambridge, MA, 1992.
19. A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
20. A. Pettorossi and M. Proietti. A Comparative Revisitation of Some Program Transformation Techniques. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 355–385. Springer LNCS 1110, 1996.
21. A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
22. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In S. Tärnlund, editor, *Proc. of Second Int'l Conf. on Logic Programming, Uppsala, Sweden*, pages 127–139, 1984.